Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic.NET

Microsoft Corporation

October 2000

Summary: This document provides recommendations for Microsoft Visual Basic developers planning to upgrade their applications to Visual Basic.NET. It includes information on the Visual Basic.NET Upgrade Tool and discusses architectural guidelines for enabling a smooth upgrade from Visual Basic 6 to Visual Basic.NET. (26 printed pages)

Contents

Overview
What Is Visual Basic.NET?
Why Is Visual Basic.NET Not 100% Compatible?
Upgrading to Visual Basic.NET
Working with Both Visual Basic 6.0 and Visual Basic.NET
Architecture Recommendations
  Browser-based Applications
  Client/Server Projects
  Single-tier Applications
  Data
Upgrading
  Variant to Object
  Integer to Short
  Property Syntax
  Visual Basic Forms to Windows Forms
  Interfaces
Upgrade Report and Comments
Programming Recommendations
  Use Early-Binding
  Use Date for Storing Dates
  Resolve Parameterless Default Properties
  Avoid Null Propagation
  Use Zero Bound Arrays
  Use Constants Instead of Underlying Values
  Arrays and Fixed-Length Strings in User-Defined Types
  Avoid Legacy Features
  Windows APIs
  Considerations for Forms and Controls

Overview

This document provides recommendations for developers using Microsoft® Visual Basic® who are planning to upgrade their applications to Microsoft Visual Basic.NET.

Visual Basic.NET will open and upgrade Visual Basic 6.0 projects to Visual Basic.NET technologies, but in most cases you will need to make some modifications to your projects after bringing them into Visual Basic.NET. The purpose of this document is to recommend how to design and implement your current Visual Basic projects to minimize the number of changes you will need to make when they are upgraded to Visual Basic.NET. Where appropriate, we use new language constructs; however, this document is not intended to be a Visual Basic.NET language reference.

> **Note** Visual Basic.NET is still in development; some compatibility details may change before the product is released. Following the guidelines in this document does not guarantee your code will not require changes; instead the guidelines aim to reduce the amount of work needed for conversion.

The upgrade wizard and command-line upgrade tools in Visual Basic.NET are still in an early stage of development and, as such, their functionality is limited. The purpose of including them in the Beta release is to give you a feel for how the upgrade process will work and to see how VB 6.0 code is modified to work in VB.NET; in Beta1, most real-world projects probably cannot be migrated successfully.

What Is Visual Basic.NET?

Visual Basic.NET is the next version of Visual Basic. Rather than simply adding some new features to Visual Basic 6.0, Microsoft has reengineered the product to make it easier than ever before to write distributed applications such as Web and enterprise n-tier systems. Visual Basic.NET has two new forms packages (Windows Forms and Web Forms); a new version of ADO for accessing disconnected data sources; and streamlined language, removing legacy keywords, improving type safety, and exposing low-level constructs that advanced developers require.

These new features open new doors for the Visual Basic developer: With Web Forms and ADO.NET, you can now rapidly develop scalable Web sites; with inheritance, the language now truly supports object-oriented programming; Windows Forms natively supports accessibility and visual inheritance; and deploying your applications is now as simple as copying your executables and components from directory to directory.

Visual Basic.NET is now fully integrated with the other Microsoft Visual Studio.NET languages. Not only can you develop application components in different programming languages, your classes can now inherit from classes written in other languages using cross-language inheritance. With the unified debugger, you can now debug multiple language applications, irrespective of whether they are running locally or on remote computers. Finally, whatever language you use, the Microsoft .NET Framework provides a rich set of APIs for Microsoft Windows® and the Internet.

Why Is Visual Basic.NET Not 100% Compatible?

There were two options to consider when designing Visual Basic.NET—retrofit the existing code base to run on top of the .NET Framework, or build from the ground up, taking full advantage of the platform. To deliver the features most requested by customers (for example, inheritance, threading), to provide full and uninhibited access to the platform, and to ensure that Visual Basic moves forward into the next generation of Web applications, the right decision was to build from the ground up on the new platform.

For example, many of the new features found in Windows Forms could have been added to the existing code base as new controls or more properties. However, this would have been at the cost of all the other great features inherent to Windows Forms, such as security and visual inheritance.

One of our major goals was to ensure Visual Basic code could fully interoperate with code written in other languages, such as Microsoft Visual C#™ or Microsoft Visual C++®, and enable the Visual Basic developer to harness the power of the .NET Framework simply, without resorting to the programming workarounds traditionally required to make Windows APIs work. Visual Basic now has the same variable types, arrays, user-defined types, classes, and interfaces as Visual C++ and any other language that targets the Common Language Runtime; however, we had to remove some features, such as fixed-length strings and non-zero based arrays from the language.

Visual Basic is now a true object-oriented language; some unintuitive and inconsistent features like **GoSub**/**Return** and **DefInt** have been removed from the language.

The result is a re-energized Visual Basic, which will continue to be the most productive tool for creating Windows-based applications, and is now positioned to be the best tool for creating the next generation Web sites.

Upgrading to Visual Basic.NET

Visual Basic.NET enables a fundamental shift from traditional Windows development to building next-generation Web and n-tier applications. For this reason, your code will need to be upgraded to take advantage of Visual Basic.NET.

This happens automatically when you open a Visual Basic 6.0 project in Visual Basic.NET: the Upgrade Wizard steps you through the upgrade process and creates a new Visual Basic.NET project (your existing project is left unchanged). This is a one-way process; the new Visual Basic.NET project cannot be opened in Visual Basic 6.0.

When your project is upgraded, the language is modified for any syntax changes and your Visual Basic 6.0 Forms are converted to Windows Forms. In most cases, you will have to make some changes to your code after it is upgraded. This is required because certain objects and language features either have no equivalent in Visual Basic.NET, or have an equivalent too dissimilar for an automatic upgrade. After the upgrade, you may also want to change your application to take advantage of some of the new features in Visual Basic.NET.

For example, Windows Forms supports control anchoring, so you can remove most of your old Visual Basic 6.0 Form resize code:

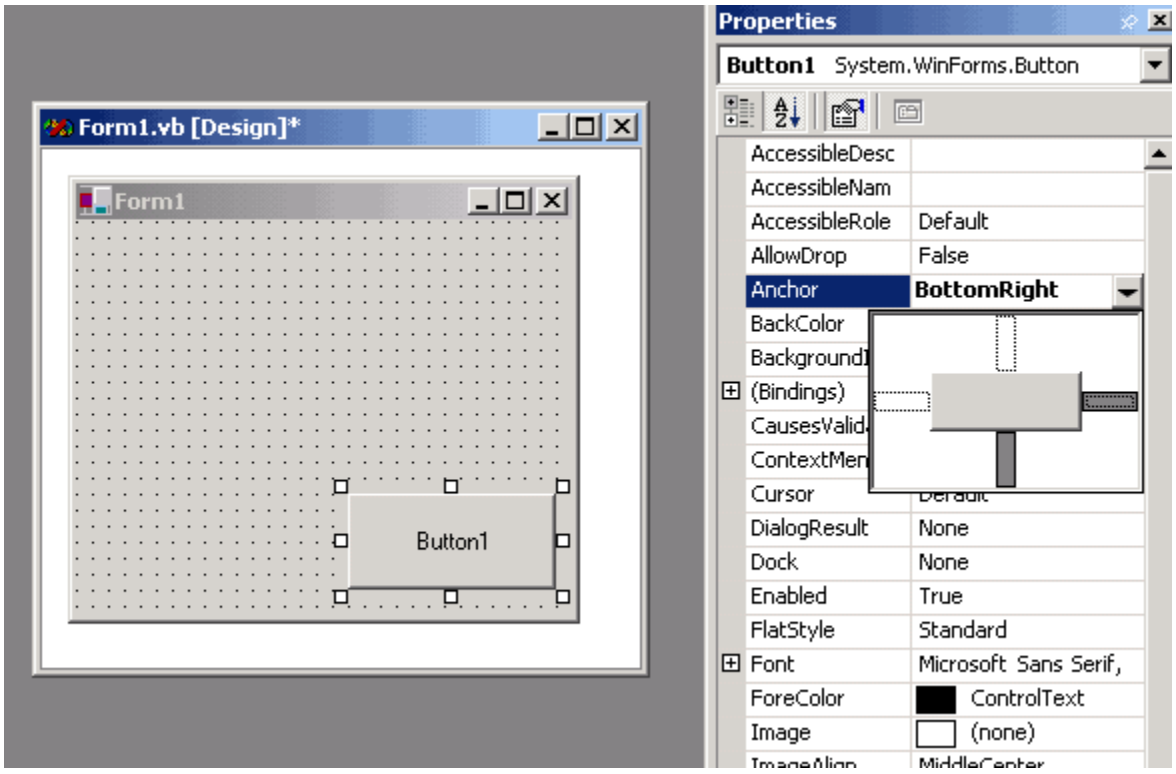

Figure 1. VB.NET support for control anchoring

To help you make the changes, after your project is upgraded, Visual Basic.NET adds an 'upgrade report' to your project itemizing any problems, and inserts comments into your upgraded code alerting you to statements that will need to be changed. Because these comments are displayed as 'TO DO' tasks in the new Task List window, you can easily see what changes are required, and navigate to the code statement simply by double-clicking the task. Each task and item in the upgrade report is associated with an online Help topic giving further guidance as to why the code needs to be changed, and what you need to do.

By following the recommendations in this document, you can minimize and, in some cases, eliminate the changes needed after upgrading your project to Visual Basic.NET. In most cases, the recommendations simply represent good programming practices; however, we also identify the

objects and methods which have no equivalents, and which should be used sparingly if you intend to upgrade your project to Visual Basic.NET.

Working with Both Visual Basic 6.0 and Visual Basic.NET

Visual Basic.NET supports upgrading Visual Basic 6.0 projects; if you have a project written in Visual Basic versions 1 to 5, we recommend you load it into VB6 (choosing to upgrade Microsoft ActiveX® controls), compile, and save the project before upgrading it to Visual Basic.NET.

Both Visual Basic.NET and Visual Basic 6.0 can be installed on the same computer and run at the same time. Likewise, applications written in Visual Basic.NET and Visual Basic 6.0 can be installed and executed on the same computer. Components written in Visual Basic.NET can interoperate with COM components written in earlier versions of Visual Basic and other languages. For example, you can drop an ActiveX control written in Visual Basic 6.0 onto a Visual Basic.NET Windows Form, use a Visual Basic 6.0 COM object from a Visual Basic.NET class library, or add a reference to a Visual Basic.NET library to a Visual Basic 6.0 executable.

Components compiled with Visual Basic.NET have subtle run-time differences from components compiled with Visual Basic 6.0. For starters, because Visual Basic.NET objects are released through garbage collection, when objects are explicitly destroyed, there may be a lag before they are actually removed from memory. There are additional differences such as the variant/object changes described later in this document. The combined result of these differences is that Visual Basic.NET applications will have similar but not identical run-time behavior to Visual Basic 6.0 applications.

In addition, Visual Basic.NET makes binary compatibility between Visual Basic.NET components and those in Visual Basic 6.0 unnecessary. Components now have a more robust versioning and deployment system than ever before, files can be deployed by simply copying to a directory (no more RegSvr32), and upgrading to a new version of a component is as simple as replacing the old file with a new file. All you have to do is ensure classes and methods are compatible with the previous version.

Architecture Recommendations

The .NET platform improves upon previous architectures, and adds greater support for scalability and distributed applications though disconnected data access, HTTP-based message transport, and file-copy based deployment (no more registering of components). To best take advantage of these features, you should design your applications with an architecture similar to that you would use in Visual Basic.NET.

Browser-based Applications

Visual Basic 6.0 and Microsoft Visual Studio® 6.0 offered several technologies for creating browser-based Internet and intranet applications:

- Webclasses
- DHTML projects
- ActiveX documents
- Active Server Pages (ASP)

Visual Basic.NET introduces ASP.NET, an enhanced version of ASP, and adds to the architecture with Web Forms, which are HTML pages with Visual Basic events. The architecture is server-based.

Below is a list of recommendations and architectural suggestions for developing Visual Basic 6.0 browser-based applications that will most seamlessly migrate to Visual Basic.NET projects:

- We recommend you use the Microsoft multi-tier architecture guidelines (see Building Solutions Today With Microsoft's Web Solution Platform)to create your applications, create the interface with ASP, and use Visual Basic 6.0 or Visual C++ 6.0 COM objects for your business logic. ASP is fully supported in Visual Basic.NET, and you can continue to extend your application using ASP, ASP.NET, and Web Forms. The Visual Basic 6.0 and Visual C++ 6.0 business objects can either be used without modification or upgraded to Visual Studio.NET.

- DHTML applications contain DHTML pages and client-side DLLs. These applications cannot be automatically upgraded to Visual Basic.NET. We recommend you leave these applications in Visual Basic 6.0.

- ActiveX documents are not supported in Visual Basic.NET, and like DHTML projects, cannot be automatically upgraded. We recommend you either leave your ActiveX document applications in Visual Basic 6.0 or, where possible, replace ActiveX documents with user controls.

- Visual Basic 6.0 ActiveX documents and DHTML applications can interoperate with Visual Basic.NET technologies. For example, you can navigate from a Visual Basic.NET Web Form to a Visual Basic 6.0 DHTML page, and vice–versa.

- Webclasses no longer exist in Visual Basic.NET. Webclass applications will be upgraded to ASP.NET; however, you will have to make some modifications after upgrading. Existing Webclass applications can interoperate with Visual Basic.NET Web Forms and ASP applications, but for new projects we recommend you use the Windows DNA platform of ASP with Visual Basic 6.0 business objects.

For more information about building applications with the Microsoft multi-tier architecture, see the Microsoft Windows DNA Web site.

Client/Server Projects

Visual Basic 6.0 offered several technologies for creating client/server applications:

- Visual Basic Forms

- Microsoft Transaction Server (MTS)/COM+ middle-tier objects

- User controls

In Visual Basic.NET, there is a new form package: Windows Forms. Windows Forms has a different object model than Visual Basic 6.0 Forms, but is largely compatible. When your project is upgraded, Visual Basic Forms are converted to Windows Forms.

Visual Basic.NET improves support for developing middle-tier MTS and COM+ component services components. Using the unified debugger, you can step from a client application into an MTS/COM+ component and back to the client. You can also use the unified debugger to step through Visual Basic 6.0 MTS/COM+ components (providing they are compiled to native code, with symbolic debug information and no optimizations).

Visual Basic.NET also introduces a new middle-tier component, Web Services. Web Services are hosted by ASP.NET, and use the HTTP transport allowing method requests to pass through firewalls. They pass and return data using industry standard XML, allowing other languages and other platforms to access their functionality. Although they do not support MTS transactions, you may

want to change your MTS/COM+ components to Web Services in cases where you do not need distributed transactions but still want to interoperate with other platforms. Although there is no automatic method for this, the task is trivial and can be completed in minutes using a drag-and-drop operation after your project has been upgraded to Visual Basic.NET.

When your project is upgraded, user controls are upgraded to Windows controls; however, custom property tag settings and accelerator keys assignments will not be upgraded.

Single-tier Applications

Visual Basic 6.0 supported building several types of single-tier applications:

- Single-tier database applications
- Visual Basic add-ins
- Utility programs and games

Single-tier database applications are typified by a Visual Basic application storing data in an Microsoft Access database. These applications will upgrade to Visual Basic.NET with some limitations (see the Data section later in this document).

Now that the Visual Basic.NET IDE is a fully integrated part of the Visual Studio.NET IDE, Visual Basic.NET has a new language-neutral extensibility model. Visual Basic.NET add-ins are now Visual Studio.NET add-ins, and you can automate and add features to any language in Visual Studio.NET. For example, you can write a Visual Basic.NET add-in that re-colors a Visual C# Windows Form or adds comments to a Visual Basic class. In order to provide this functionality, Visual Basic.NET has moved away from the old extensibility model, and you will need to change the extensibility objects in your application to take advantage of the new features.

Many applications fall under the category of Utility programs. Utility applications that manipulate files, registry settings, and the like will often upgrade without requiring any additional changes. After upgrading, there are many new features you can take advantage of, such as exception handling in the language to capture file system errors, and using .NET Framework registry classes to manipulate the registry. One thing to be aware of is that applications relying on specific performance characteristics of Visual Basic 6.0, such as arcade games, will probably require some modifications because Visual Basic.NET has different performance characteristics. For games support in Visual Basic.NET, you can use Microsoft DirectX® 7, or the new version of GDI. GDI+ introduces many new features, including Alpha blending support for all 2-D graphics primitives, anti-aliasing, and expanded support for image file formats.

Data

Visual Basic 6.0 offered several types of data access:

- ActiveX Data Objects (ADO)
- Remote Data Objects (RDO)
- Data Access Objects (DAO)

Visual Basic.NET introduces an enhanced version of ADO called ADO.NET. ADO.NET targets disconnected data, and provides performance improvements over ADO when used in distributed applications. ADO.NET offers read/write data binding to controls for Windows Forms and read-only data binding for Web Forms.

DAO, RDO, and ADO can still be used in code from Visual Basic.NET, with some trivial modifications (covered in the language section of this document). However, Visual Basic.NET does not support

DAO and RDO data binding to controls, data controls, or RDO User connection. We recommend that if your applications contain DAO or RDO data binding you either leave them in Visual Basic 6.0 or upgrade the DAO and RDO data binding to ADO before upgrading your project to Visual Basic.NET, as ADO data binding is supported in Windows Forms. Information on how to do this is available in the Visual Basic 6.0 Help.

In summary, we recommend using ADO in your Visual Basic 6.0 projects.

Upgrading

When your code is upgraded, Visual Basic.NET creates a new upgraded project and makes most of the required language and object changes for you. The following sections provide a few examples of how your code is upgraded.

Variant to Object

Previous versions of Visual Basic supported the **Variant** datatype, which could be assigned to any primitive type (except fixed-length strings), Empty, Error, Nothing and Null. In Visual Basic.NET, the functionality of the **Variant** and **Object** datatypes is combined into one new datatype: **Object**. The **Object** datatype can be assigned to primitive datatypes, Empty, Nothing, Null, and as a pointer to an object.

When your project is upgraded to Visual Basic.NET, all variables declared as **Variant** are changed to **Object**. Also, when code is inserted into the editor, the **Variant** keyword is replaced with **Object**.

Integer to Short

In Visual Basic.NET, the datatype for 16-bit whole numbers is now **Short**, and the datatype for 32-bit whole numbers is now **Integer** (**Long** is now 64 bits). When your project is upgraded, the variable types are changed:

```
Dim x As Integer
dim y as Long
```

is upgraded to:

```
Dim x As Short
dim y as Integer
```

Property Syntax

Visual Basic.NET introduces a more intuitive syntax for properties, which groups **Get** and **Set** together. Your property statements are upgraded as shown in the following example:

```
Property Get MyProperty() As Integer
    MyProperty = m_MyProperty
End Property
Property Let MyProperty(NewValue As Integer)
    m_MyProperty = NewValue
End Property
```

is upgraded to:

```
Property MyProperty() As Short
```

```
    Get
        MyProperty = m_MyProperty
    End Get
    Set
        m_MyProperty = Value
    End Set
End Property
```

Visual Basic Forms to Windows Forms

Visual Basic.NET has a new forms package, Windows Forms, which has native support for accessibility and has an in-place menu editor. Your existing Visual Basic Forms are upgraded to Windows Forms.

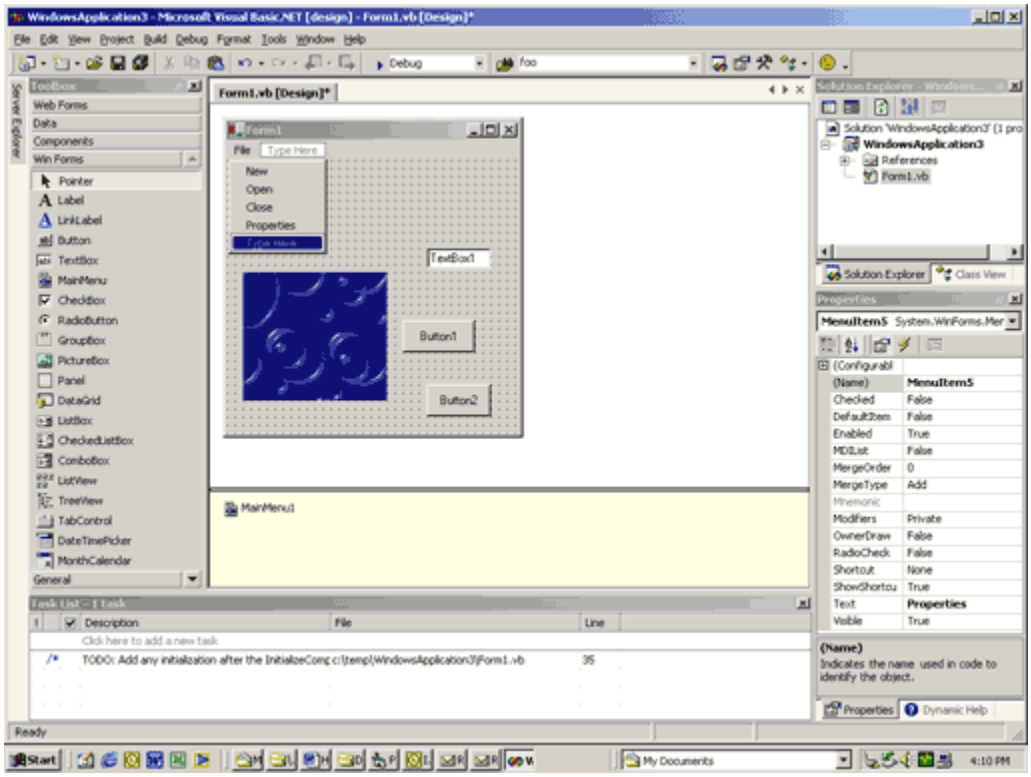

Figure 2. Windows Forms in-place menu editor. (Click figure to see larger image.)

Interfaces

In previous versions of Visual Basic, interfaces for public classes were always hidden from the user. In Visual Basic.NET, they can be viewed and edited in the Code Editor. When your project is upgraded, you choose whether to have interface declarations automatically created for your public classes.

Upgrade Report and Comments

After your project is upgraded, an upgrade report is added to your project, itemizing any changes you will need to make to your upgraded code. Additionally, comments are added to your code to alert you to any potential problems. These comments show up automatically in the Visual Studio.NET Task List.

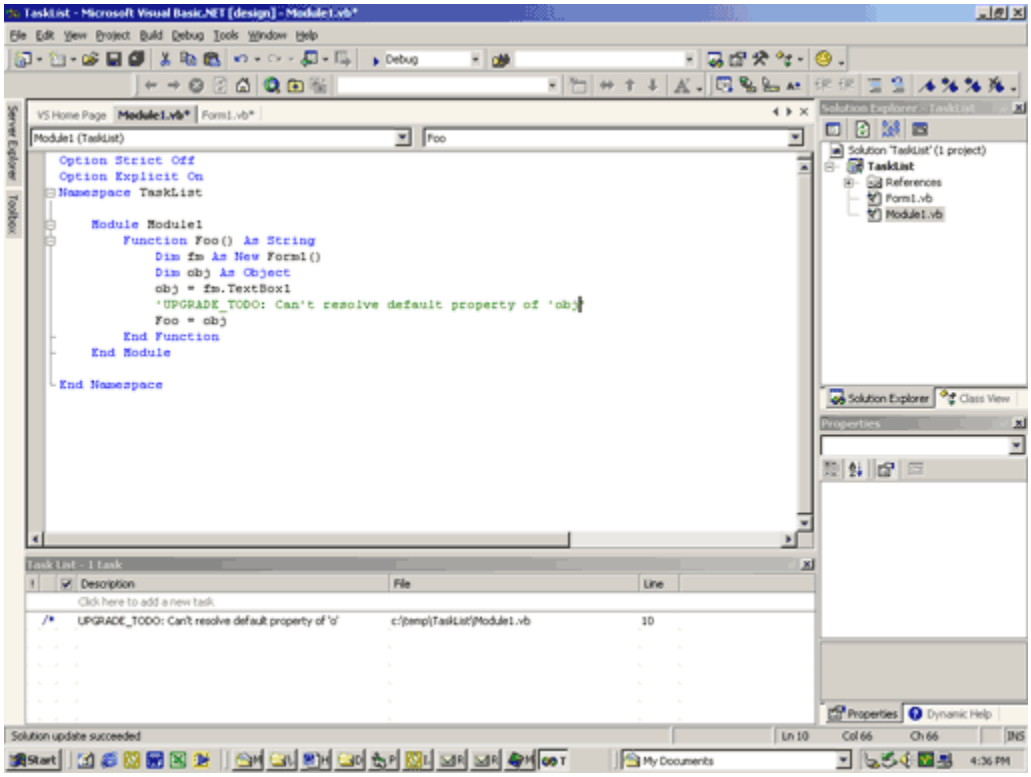

Figure 3. Upgrade comments are added to Visual Basic code as well as the Task List. (Click figure to see larger image.)

Programming Recommendations

This section provides recommendations for how you should write code to minimize the changes you will need to make after upgrading your project to Visual Basic.NET.

Use Early-Binding

Both Visual Basic 6.0 and Visual Basic.NET support late-bound objects, which is the practice of declaring a variable as the **Object** datatype and assigning it to an instance of a class at run time. However, during the upgrade process, late-bound objects can introduce problems when resolving default properties, or in cases where the underlying object model has changed and properties, methods, and events need to be converted. For example, suppose you have a Form called Form1 with a label called Label1; the following Visual Basic 6.0 code would set the caption of the label to "SomeText":

```
Dim o As Object
Set o = Me.Label1
o.Caption = "SomeText"
```

In Visual Basic.NET Windows Forms, the **Caption** property of a label control is now called **Text**. When your code is upgraded, all instances of the **Caption** property are changed to **Text**, but because a late-bound object is type-less, Visual Basic cannot detect what type of object it is, or if any properties should be translated. In such cases, you will need to change the code yourself after upgrading.

If you rewrite the code using early-bound objects, it will be upgraded automatically:

```
Dim o As Label
Set o = Me.Label1
o.Caption = "SomeText"
```

Where possible you should declare variables of the appropriate object type rather than simply declaring them as the **Object** datatype.

In the cases where you do use Object and Variant variables in your Visual Basic 6.0 code, we recommend you use explicit conversions when you assign the variables, perform operations on the variables, or pass the variables to a function. For example, the intention of the '+' operation in the following code is unclear:

```
Dim Var1 As Variant
Dim Var2 As Variant
Dim Var3 As Variant
Var1 = "3"
Var2 = 4
Var3 = Var1 + Var2   'UNCLEAR: What is the intention?
```

Should Var1 and Var2 be added as strings or integers?

The above example may result in a run-time error in Visual Basic.NET. Rewriting the final line to use explicit conversions ensures the code will work:

```
Var3 = CInt(Var1) + CInt(Var2)   'GOOD: explicit conversion
```

Visual Basic.NET supports overloading functions based on parameter type. For example, the **Environ** function now has two forms:

```
Environ( Expression As Integer) As String
Environ( Expression As String ) As String
```

Visual Basic.NET determines which function to call based on the parameter type. If you pass an integer to **Environ()**, the integer version is called; if you pass a string, then the string version is called. Code that passes a **Variant** or **Object** datatype to an overloaded function may cause a compile or runtime error. Using an explicit conversion, as in the following example, will mean your code will work as intended after it is upgraded to Visual Basic.NET:

```
Dim a As String
Dim v As Variant
v = "Path"
a = Environ(CStr(v))   'GOOD: explicit conversion
```

Using explicit conversions of late bound objects is good coding practice. It makes the intention of the code easy to determine, and makes it easier for you to move your project to Visual Basic.NET.

Use Date for Storing Dates

Earlier versions of Visual Basic supported using the **Double** datatype to store and manipulate dates. You should not do this in Visual Basic.NET, because dates are not internally stored as doubles. For example, the following is valid in Visual Basic 6.0, but may cause a compile error in Visual Basic.NET:

```
Dim dbl As Double
```

```
Dim dat As Date
dat = Now
dbl = dat      'VB.NET: Double can't be assigned to a date
dbl = DateAdd("d", 1, dbl)   'VB.NET: Can't use Double in date
functions
dat = CDate(dbl)   'VB.NET: CDate can't convert double to date
```

The .NET framework provides the **ToOADate** and **FromOADate** functions to convert between doubles and dates. However, when your project is upgraded to Visual Basic.NET, it is difficult to determine the intention of code that uses doubles to store dates. To avoid unnecessary modifications to your code in Visual Basic.NET, always use the **Date** datatype to store dates.

Resolve Parameterless Default Properties

In Visual Basic 6.0, many objects expose default properties, which can be omitted as a programming shortcut. For example, **TextBox** has a default property of **Text**, so instead of writing:

```
MsgBox Form1.Text1.Text
```

you use the shortcut:

```
MsgBox Form1.Text1
```

The default property is resolved when the code is compiled. In addition, you could also use default properties with late-bound objects, as in the following example:

```
Dim obj As Object
Set obj = Form1.Text1
MsgBox obj
```

In the late-bound example, the default property is resolved at run time, and the **MsgBox** displays the value of the default property of the **TextBox** as **Text1**.

Visual Basic.NET does not support parameterless default properties, and consequently does not allow this programming shortcut. When your project is upgraded, Visual Basic.NET resolves the parameterless default properties, but late-bound usages that rely on run-time resolution cannot be automatically resolved. In these cases, you will have to change the code yourself. An additional complication is that many libraries implement default properties using a property called **_Default**. **_Default** acts as a proxy, passing calls to the real default property. So, when your project is upgraded, some default properties will be resolved to **_Default**. The code will still work as usual, but it will be less understandable than code written explicitly using the actual property. For these reasons, try to avoid using parameterless default properties in your Visual Basic 6.0 code. Instead of writing:

```
Dim obj As Object
Set obj = Me.Text1
MsgBox obj   'Relying on default property
MsgBox Me.Text1   'Relying on default property
```

use:

```
Dim obj As Object
Set obj = Me.Text1
MsgBox obj.Text   'GOOD: Default property is resolved
```

```
MsgBox Me.Text1.Text   'GOOD: Default property is resolved
```

While *parameterless* default properties are not supported in Visual Basic.NET, default properties *with parameters* are supported. To understand the difference between the two types, consider that parametered default properties always have an index. An example is the default property of ADO **recordset**: the **Fields** collection. The code:

```
Dim rs As ADODB.Recordset
rs("CompanyName") = "SomeCompany"
rs!CompanyName = "SomeCompany"
```

is actually a shortcut for:

```
Dim rs As ADODB.Recordset
rs.Fields("CompanyName").Value = "SomeCompany"
rs.Fields!CompanyName.Value = "SomeCompany"
```

In this case, the **Fields** property is parametered, and so the usage is valid in Visual Basic.NET; however, the default property of the **Fields** property, **Value,** is parameterless, so the correct usage in Visual Basic.NET is:

```
Dim rs As ADODB.Recordset
rs("CompanyName").Value = "SomeCompany"
rs!CompanyName.Value = "SomeCompany"
```

This example and most other default properties are resolved for you when the project is upgraded, so resolving them in Visual Basic 6.0 is simply a good programming practice. However, you should avoid using default properties with the **Object** and **Variant** datatypes, as these cannot be resolved and you will have to fix the code yourself in the upgraded project.

Avoid Null Propagation

Previous versions of Visual Basic supported Null propagation. Null propagation supports the premise that when null is used in an expression, the result of the expression will itself be Null. In each case in the following example, the result of V is always Null.

```
Dim V
V = 1 + Null
V = Null + Right$("SomeText", 1)
V = Right("SomeText", 0)
```

Null propagation is not supported in Visual Basic.NET. The statement **1+Null** will generate a type mismatch in Visual Basic.NET. Additionally, where Visual Basic 6.0 had two versions of the **Left** function—**Left$** returning a string, **Left** returning a variant which could be Null—Visual Basic.NET only has one version, **Left**, which always returns a string.

In order to be compatible with both Visual Basic 6.0 and Visual Basic.NET you should always write code to test for Null instead of relying on Null propagation. Furthermore, in Visual Basic.NET, the following functions will no longer return Null:

Chr	**Mid**
Command	**Oct**

CurDir	**Right**
Date	**RTrim**
Environ	**Space**
Error	**Str**
Hex	**Time**
LCase	**Trim**
LTrim	**UCase**

Null propagation is commonly used in database applications, where you need to check if a database field contains Null. In these cases you should check results using the function **IsNull()** and perform the appropriate action.

A related issue involves concatenating a string with a Null value. When programming with database objects, it is common practice to concatenate an "empty string" to a field to ensure that Null values are coerced to an empty string. For example:

```
MyString = rs!Field1 & ""
```

This technique is still supported in Visual Basic.NET. When a Null value is concatenated with an empty string (using the & operator), the result is an empty string.

Use Zero Bound Arrays

Visual Basic 6.0 allowed you to define arrays with lower and upper bounds of any whole number. You could also use **ReDim** to reassign a variant as an array. To enable interoperability with other languages, arrays in Visual Basic.NET must have a lower bound of zero, and **ReDim** cannot be used unless the variable was previously declared with Dim As Array. Although this restricts the way arrays can be defined, it does allow you to pass arrays between Visual Basic.NET and any other .NET language. The following example shows the restriction:

```
Dim a(1 To 10) As Integer   'LBound must be 0 in VB.NET
Dim v
ReDim v(10)   'Can't use ReDim without Dim in VB.NET
Dim b(10) As Integer   'GOOD: Creates an array of 10 integers
ReDim b(5) As Integer   'GOOD: Can ReDim previously Dimed var
```

A side effect is that Option Base 0|1 is removed from the language.

When your project is upgraded to Visual Basic.NET, any option base statements are removed from your code. If the array is zero bound, it is left unchanged. However, if an array is non-zero bound, then the lower bound is removed and a warning is inserted into the code, as in the following example:

```
Dim a(1 To 10) As Integer
```

changes to:

```
'UPGRADE_WARNING: Lower Bound of array a was changed from 1 to 0
Dim a(10) As Integer
```

In many cases, the upgraded code will work as it did before. However, if your application logic relies on the lower bound being 1, then you will need to make some modifications. **Dim, ReDim**, and **LBound** statements are marked with warnings to help you review the changes.

For this reason, you should use zero bound arrays in your Visual Basic 6.0 code, avoid using **ReDim** as an array declaration, and avoid using Option Base 1.

Use Constants Instead of Underlying Values

When writing code, try to use constants rather than relying on their underlying values. For example, if you are maximizing a form at run time, use:

```
Me.WindowState = vbMaximized   'Good: Constant name used
```

rather than:

```
Me.WindowStyle = 2   'Avoid using underlying value
Me.WindowStyle = X   'Avoid using variables
```

Likewise, use **True** and **False** instead of -1 and 0.

In Visual Basic.NET, the values and in some cases the names of some properties and constants have changed; for example, the value of **True** changes from -1 to 1. When your project is upgraded to Visual Basic.NET, most constants are changed automatically for you; however, if you use underlying values or variables instead of the constant names, many cases cannot be upgraded automatically. Using constant names minimizes the number of modifications you have to do.

Arrays and Fixed-Length Strings in User-Defined Types

Due to changes made which allow Visual Basic.NET arrays and structures to be fully compatible with other Visual Studio.NET languages, fixed-length strings are no longer supported in the language. In most cases this is not a problem, because there is a compatibility class which provides fixed-length string behavior, so the code:

```
Dim MyFixedLengthString As String * 100
```

upgrades to the following:

```
Dim MyFixedLengthString As New VB6.FixedLengthString(100)
```

However, fixed-length strings do cause a problem when used in structures (also known as user-defined types). The problem arises because the fixed-length string class is not automatically created when the user-defined type is created. An additional problem is that fixed-size arrays are not created, either, when the user-defined type is created.

When your code is upgraded, user-defined types with fixed-length strings or arrays will be marked with a comment telling you to initialize the fixed-length string or array before using the user-defined type. However, you can shield yourself from this modification by changing your Visual Basic 6.0 user-defined types to use strings instead of fixed-length strings, and uninitialized arrays instead of fixed-size arrays. For example:

```
Private Type MyType
```

```
    MyArray(5) As Integer
    MyFixedString As String * 100
End Type
Sub Bar()
    Dim MyVariable As MyType
End Sub
```

can be changed to:

```
Private Type MyType
    MyArray() As Integer
    MyFixedString As String
End Type
Sub Bar()
    Dim MyVariable As MyType
    ReDim MyVariable.MyArray(5) As Integer
    MyVariable.MyFixedString = String$(100, " ")
End Sub
```

Avoid Legacy Features

Because they have been removed from the language, you should avoid using the following keywords:

- **Def<type>**
- **Computed GoTo/GoSub**
- **GoSub/Return**
- **Option Base 0|1**
- **VarPtr, ObjPtr, StrPtr**
- **LSet**

These are explained in more detail below.

Def<type>

In previous versions of Visual Basic, DefBool, DefByte, DefInt, DefLng, DefCur, DefSng, DefDbl, DefDec, DefDate, DefStr, DefObj and DefVar were used in the declarations section of a module to define a range of variables as a certain type. For example:

```
DefInt A-C
```

defined all variables beginning with the letter A, B, or C as an integer. Instead of using **Def<type>** statements, you should explicitly declare variables.

Computed GoTo/GoSub

Computed GoTo/GoSub statements take this form:

```
On x GoTo 100, 200, 300
```

These are not supported in Visual Basic.NET. Instead, you should use **If** statements, and Select Case constructs.

GoSub/Return

GoSub and **Return** statements are not supported in Visual Basic.NET. In most cases you can replace these with functions and procedures.

Option Base 0|1

Option Base 0|1 was used to specify the default lower bound of an array. As mentioned previously, this statement has been removed from the language since Visual Basic.NET natively only supports arrays with a zero lower bound. Non-zero lower bound arrays are supported through a wrapper class.

VarPtr, ObjPtr, StrPtr

VarPtr, **VarPrtArray**, **VarPtrStringArray**, **ObjPtr** and **StrPtr** were undocumented functions used to get the underlying memory address of variables. These functions are not supported in Visual Basic.NET.

LSet

In Visual Basic 6.0, the **LSet** statement could be used to assign a variable of one user-defined type to another variable of a different user-defined type. This functionality is not supported in Visual Basic.NET.

Windows APIs

Many APIs can be used exactly as they were in Visual Basic 6.0, with the caveat that you have to adjust your data types accordingly. The Visual Basic 6.0 **Long** datatype is now the Visual Basic.NET **Integer** datatype, and the Visual Basic 6.0 **Integer** datatype is now the Visual Basic.NET **Short** datatype. During the upgrade, these changes are made for you, and simple APIs work exactly the same as they did in Visual Basic 6.0. For example:

```
Private Declare Function GetVersion Lib "kernel32" () As Long
Function GetVer()
    Dim Ver As Long
    Ver = GetVersion()
    MsgBox ("System Version is " & Ver)
End Function
```

changes to:

```
Private Declare Function GetVersion Lib "kernel32" () As Integer
Function GetVer()
    Dim Ver As Integer
    Ver = GetVersion()
    MsgBox("System Version is " & Ver)
End Function
```

In addition to numeric datatype upgrades, Visual Basic 6.0 had a fixed-length string data type which is not supported in Visual Basic.NET, and which is upgraded to a fixed-length string wrapper class. In many cases in Visual Basic 6.0 you can perform the same action using a normal string. For example:

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Long) As
Long
Function GetUser()
    Dim Ret As Long
    Dim UserName As String
    Dim Buffer As String * 25
    Ret = GetUserName(Buffer, 25)
    UserName = Left$(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox (UserName)
End Function
```

can be better written using a normal string explicitly set to length 25 instead of a fixed-length string:

```
    Dim Buffer As String
    Buffer = String$(25, " ")
```

This is upgraded to Visual Basic.NET as follows:

```
Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Integer)
As Integer
Function GetUser()
    Dim Ret As Integer
    Dim UserName As String
    Dim Buffer As String
    Buffer = New String(CChar(" "), 25)
    Ret = GetUserName(Buffer, 25)
    UserName = Left(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox(UserName)
End Function
```

In some cases, Visual Basic.NET better handles passing strings to APIs, since you can optionally declare how you want strings to be passed using the **ANSI** and **UNICODE** keywords.

There are three cases where you may need to make some changes. The first is passing user-defined types that contain fixed-length strings or byte arrays to APIs. In Visual Basic.NET you may need to change your code, adding the **MarshallAs** attribute (from **System.Runtime.InteropServices**) to each fixed-length string or byte array in the user-defined type. The second case is using the **As Any** variable type in a **Declare** statement. This is not supported in Visual Basic.NET. Variables of type **As Any** were often used to pass a variable that was either a string or Null; you can replace this Visual Basic 6.0 usage by declaring two forms of the API, one with longs, one with strings. For example, the GetPrivateProfileString API has a parameter *lpKeyName* of type **As Any**:

```
Private Declare Function GetPrivateProfileString Lib "kernel32"
Alias
   "GetPrivateProfileStringA" (ByVal lpApplicationName As String,
ByVal
      lpKeyName As Any, ByVal lpDefault As String, ByVal
         lpReturnedString As String, ByVal nSize As Long, ByVal
            lpFileName As String) As Long
```

You can remove the “As Any” by replacing the **Declare** with two versions; one that accepts a long, and one that accepts a string:

```
Private Declare Function GetPrivateProfileStringKey Lib
"kernel32" Alias
   "GetPrivateProfileStringA" (ByVal lpApplicationName As String,
ByVal
      lpKeyName As String, ByVal lpDefault As String, ByVal
         lpReturnedString As String, ByVal nSize As Long, ByVal
            lpFileName As String) As Long
Private Declare Function GetPrivateProfileStringNullKey Lib
"kernel32"
   Alias "GetPrivateProfileStringA" (ByVal lpApplicationName As
String,
      ByVal lpKeyName As Long, ByVal lpDefault As String, ByVal
         lpReturnedString As String, ByVal nSize As Long, ByVal
            lpFileName As String) As Long
```

When you wish to pass the value Null to the API, you use the **GetPrivateProfileStringNullKey** version. Doing it this way means that the function upgrades to Visual Basic.NET.

The final area where you may need to make some changes is if you are using APIs that perform thread creation, Windows subclassing, message queue hooking, and so on. Some of these functions will cause a run-time error in Visual Basic.NET. Many of these APIs have equivalents in Visual Basic.NET or the .NET Framework. You will have to fix these on a case-by-case basis.

Considerations for Forms and Controls

Visual Basic.NET has a new forms package, Windows Forms. Windows Forms is largely compatible with the forms package found in Visual Basic 6; however, there are some key differences that are outlined below:

- Windows Forms does not support the OLE container control; you should avoid using this control in your Visual Basic 6.0 applications.
- There is no shape control in Windows Forms. Square and rectangular shapes will be upgraded to labels, while ovals and circles cannot be upgraded. You should avoid using these in your applications.
- There is no line control in Windows Forms. Horizontal and vertical lines are upgraded to labels. Diagonal lines are not upgraded, and you should avoid using them.
- Windows Forms has a new set of graphics commands that replace the Form methods **Circle**, **CLS**, **PSet**, **Line**, and **Point**. Because the new object model is quite different from Visual Basic 6.0, these methods cannot be upgraded.
- For the Timer control, setting the **Interval** property to 0 does not disable the timer; instead the interval is reset to 1. In your Visual Basic 6.0 projects, you should set **Enabled** to **False** instead of setting the **Interval** to 0.
- Windows Forms has two menu controls, MainMenu and ContextMenu, whereas Visual Basic 6.0 has one menu control, Menu, which can be opened as a MainMenu or a ContextMenu. Menu controls are upgraded to MainMenu

controls, but you will not be able to use them as ContextMenus; you will have to recreate your ContextMenus.

- Windows Forms has no support for Dynamic Data Exchange (DDE).
- Windows Forms does not support the **Form.PrintForm** method.
- Although Windows Forms has support for drag-and-drop functionality, the object model is quite different from Visual Basic 6.0. Therefore, the Visual Basic 6.0 drag-and-drop properties and methods cannot be upgraded.
- The .NET framework has an improved Clipboard object (**System.Windows.Clipboard**) that offers more functionality and supports more clipboard formats than the Visual Basic 6.0 Clipboard object. However, because of differences between object models, clipboard statements cannot be automatically upgraded.
- To ensure your forms are upgraded to the right size, you should always use the default ScaleMode of twips in your applications. During the upgrade, Visual Basic.NET transforms your forms coordinates from twips to pixels.
- Windows Forms only supports true-type and open-type fonts. If your application uses other fonts, these fonts will be changed to the system's default font, and all formatting (size, bold, italic, underline) will be lost. This applies to the default VB6 font MS Sans Serif. For this reason, we recommend you use Arial instead of MS Sans Serif, wherever you have formatted text.

Send feedback on this article. Find support options.

© 2001 Microsoft Corporation. All rights reserved. Terms of use.